

LETTER

Exception Handling in Logic Programming

Keehang KWON[†], *Member*

SUMMARY One of the long-standing problems on logic programming is to express exception handling in a high-level way. We argue that this problem can be solved by adopting computability logic and sequential-choice disjunctive goal formulas of the form $G_0 \nabla G_1$ where G_0, G_1 are goals. These goals have the following intended semantics: sequentially *choose* the first true goal G_i and execute G_i where $i (= 0 \text{ or } 1)$. These goals thus allow us to specify a task G_0 with the failure-handling (exception handling) routine G_1 .

key words: *Prolog, exception handling, failure handling, computability logic*

1. Introduction

One of the major problems on logic programming is to treat the extra-logical primitive in a *high-level* way. The progress of logic programming has enriched the theory of Horn clauses with higher-order programming, mutual exclusion, etc. Nevertheless exception handling could not be dealt with elegantly.

In this paper, we propose a purely logical solution to this problem. It involves the direct employment of computability logic (CL) [3] to allow for goals with exception handling capability. A sequential-choice disjunctive (SCD) goal – is of the form $G_0 \nabla G_1$ where G_0, G_1 are goals. Executing this goal with respect to a program $D - ex(D, G_0 \nabla G_1)$ – has the following intended semantics:

sequentially choose the first true one between

$$ex(D, G_0), ex(D, G_1).$$

An illustration of this aspect is provided by the following definition of the relation $sort(X, Y)$ which holds if Y is a sorted list of X :

$$sort(X, Y) : - \text{heapsort}(X, Y) \nabla \text{quicksort}(X, Y)$$

The body of the definition above contains a SCD goal, denoted by ∇ . As a particular example, solving the query $sort([3, 100, 40, 2], Y)$ would result in selecting and executing the first goal $\text{heapsort}([3, 100, 40, 2], Y)$. If the heapsort module is available in the program, then the given goal will succeed, producing solutions for Y .

If the execution fails, the machine tries the plan B , *i.e.*, the quicksort module.

As seen from the example above, SCD goals of the form $A \nabla B$ can be used to specify a task A , together with the failure-handling routine B .

This paper proposes Prolog[∇], an extension of Prolog with SCD operators in goal formulas. The remainder of this paper is structured as follows. We describe Prolog[∇] in the next section. In Section 3, we present some examples of Prolog[∇]. Section 4 concludes the paper.

2. The Language

The language is a version of Horn clauses with SCD goals. It is described by G - and D -formulas given by the syntax rules below:

$$G ::= A \mid G \wedge G \mid \exists x G \mid G \nabla G$$

$$D ::= A \mid G \supset A \mid \forall x D \mid D \wedge D$$

In the rules above, A represents an atomic formula. A D -formula is called a Horn clause with SCD goals.

We will present a machine's strategy for this language as a set of rules. These rules in fact depend on the top-level constructor in the expression, a property known as uniform provability[6]–[8]. Note that execution alternates between two phases: the goal-reduction phase and the backchaining phase. In the goal-reduction phase (denoted by $ex(D, G)$), the machine tries to solve a goal G from a clause D by simplifying G . The rule (6) – (8) are related to this phase. If G becomes an atom, the machine switches to the backchaining mode. This is encoded in the rule (5). In the backchaining mode (denoted by $bc(D_1, D, A)$), the machine tries to solve an atomic goal A by first reducing a Horn clause D_1 to simpler forms (via rule (3) and (4)) and then backchaining on the resulting clause (via rule (1) and (2)).

Definition 1. Let G be a goal and let D be a program. Then the notion of executing $\langle D, G \rangle - ex(D, G)$ – is defined as follows:

- (1) $bc(A, D, A)$. % This is a success.

Manuscript received January 1, 2003.

Manuscript revised January 1, 2003.

Final manuscript received January 1, 2003.

[†]The author is a professor of Computer Eng., DongA University. email:khkwon@dau.ac.kr

- (2) $bc((G_0 \supset A), D, A)$ if $ex(D, G_0)$.
- (3) $bc(D_1 \wedge D_2, D, A)$ if $bc(D_1, D, A)$ or $bc(D_2, D, A)$.
- (4) $bc(\forall x D_1, D, A)$ if $bc([t/x]D_1, D, A)$.
- (5) $ex(D, A)$ if $bc(D, D, A)$.
- (6) $ex(D, G_0 \wedge G_1)$ if $ex(D, G_0)$ and $ex(D, G_1)$.
- (7) $ex(D, \exists x G_0)$ if $ex(D, [t/x]G_0)$.
- (8) $ex(D, G_0 \nabla G_1)$ if select the first successful disjunct between $ex(D, G_0)$ and $ex(D, G_1)$. % This goal behaves as a goal with exception handling.

In the above rules, only the rule (8) is a novel feature. To be specific, this goal first attempts to execute G_0 . If it succeeds, then do nothing (and do not leave any choice point for G_1). If it fails, then G_1 is attempted.

3. Examples

As an example, let us consider the following database which contains the today's flight information for major airlines such as Panam and Delta airlines.

```
% panam(source, destination, dp_time, ar_time)
% delta(source, destination, dp_time, ar_time)
panam(paris, nice, 9 : 40, 10 : 50)
:
panam(nice, london, 9 : 45, 10 : 10)
delta(paris, nice, 8 : 40, 09 : 35)
:
delta(paris, london, 9 : 24, 09 : 50)
```

Consider a goal $\exists dt \exists at \text{ panam}(\text{paris}, \text{london}, dt, at) \nabla \exists dt \exists at \text{ delta}(\text{paris}, \text{london}, dt, at)$. This goal expresses the task of finding whether the user has a flight in Panam to fly from paris to london today. Since there is no Panam flight, the system now tries Delta. Since Delta has a flight, the system produces the departure and arrival time of the flight of the Delta airline.

4. Conclusion

In this paper, we have considered an extension to Prolog with SCD goals. This extension allows goals of the form $G_0 \nabla G_1$ where G_0, G_1 are goals. These goals are particularly useful for specifying exception handling in Prolog, making Prolog more versatile.

In the near future, we plan to investigate the connection between our execution model and Japaridze's Computability Logic (CL) [3], [4]. CL is a new semantic platform for reinterpreting logic as a theory of tasks. Formulas in CL stand for instructions that can carry out some tasks. We plan to investigate whether our operational semantics is sound and complete with respect to the semantics of CL.

5. Acknowledgements

This work was supported by Dong-A University Research Fund.

References

- [1] I. Bratko, "Prolog: programming for AI", Addison Wesley, 2001 (3rd edition).
- [2] J. Hodas and D. Miller, "Logic Programming in a Fragment of Intuitionistic Linear Logic", Information and Computation, vol.110, pp.327–365, 1994.
- [3] G. Japaridze, "Introduction to computability logic", Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.
- [4] G. Japaridze, "Sequential operators in computability logic", Information and Computation, vol.206, No.12, pp.1443–1475, 2008.
- [5] J. Kriener and A. King, "RedAlert: Determinacy Inference for Prolog", Theory and Practice of Logic Programming, vol.11, no.4-5, pp.182–196.
- [6] E. Komendantskaya and V. Komendantsky, "On uniform proof-theoretical operational semantics for logic programming", In J.-Y. Beziau and A. Costa-Leite, editors, Perspectives on Universal Logic, pages 379–394. Polimetrica Publisher, 2007.
- [7] D. Miller, "A logical analysis of modules in logic programming", Journal of Logic Programming, vol.6, pp.79–108, 1989.
- [8] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, "Uniform proofs as a foundation for logic programming", Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.